下载APP

# 09｜即学即练：构建一个Web服务就是这么简单

2021-11-01 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

**讲述：Tony Bai**
时长 20:02 大小 18.35M

你好，我是 Tony Bai。

在入门篇前面的几节课中，我们已经从 Go 开发环境的安装，一路讲到了 Go 包的初始化次序与 Go 入口函数。讲解这些，不仅仅是因为它们是你学习 Go 语言的基础，同时我想为你建立“手勤”的意识打好基础。

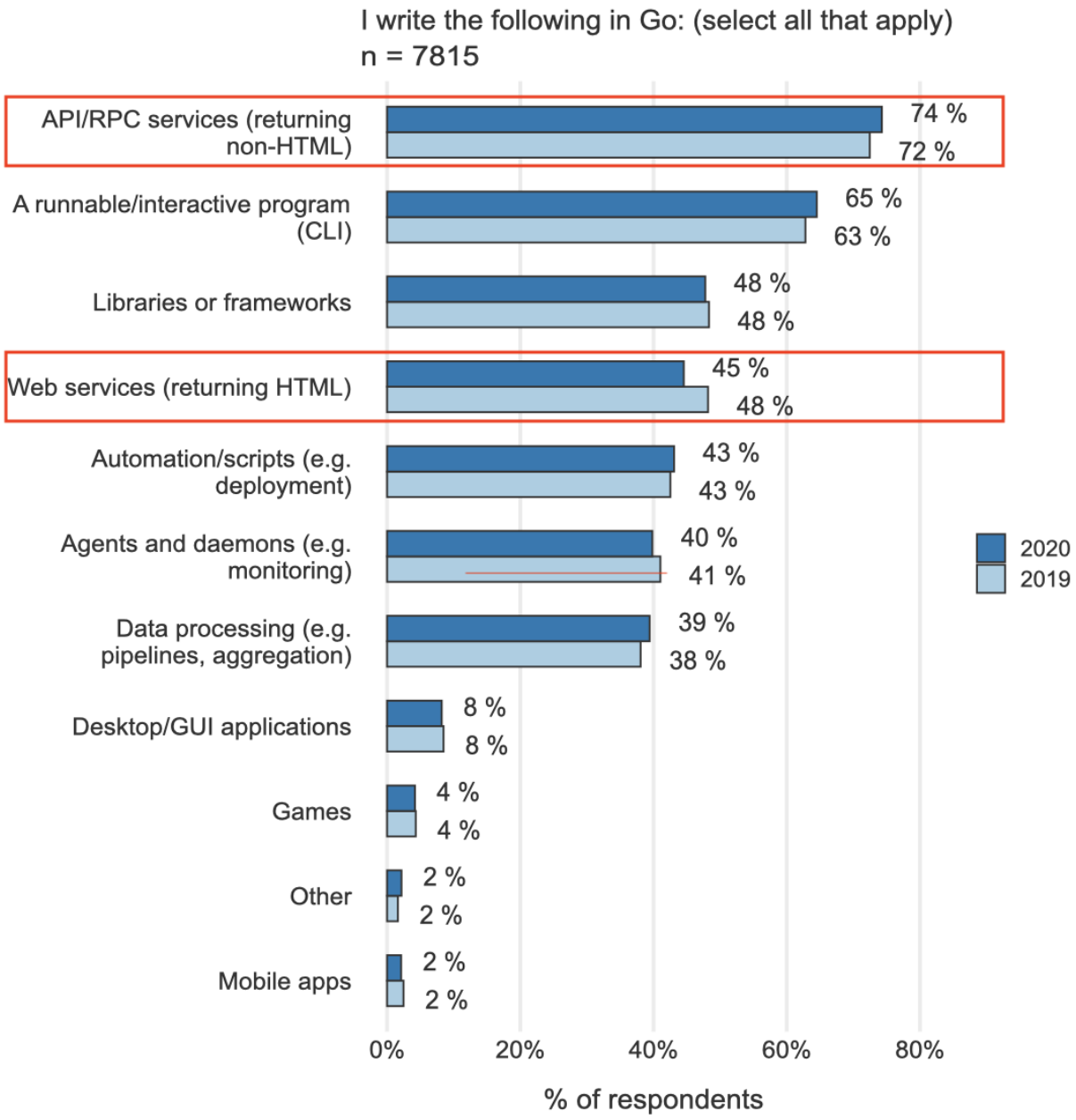
作为 Go 语言学习的“过来人”，学到这个阶段，我深知你心里都在跃跃欲试，想将前面学到的知识综合运用起来，实现一个属于自己的 Go 程序。但到目前为止，我们还没有开始 Go 基础语法的系统学习，你肯定会有一种“无米下炊”的感觉。

领资料

不用担心，我在这节课安排了一个实战小项目。在这个小项目里，我希望你不要困在各种语法里，而是先跟着我“照猫画虎”地写一遍、跑一次，感受 Go 项目的结构，体会 Go 语言的魅力。

## 预热：最简单的 HTTP 服务

在想选择以什么类型的项目的时候，我还颇费了一番脑筋。我查阅了Go 官方用户 2020 调查报告，找到 Go 应用最广泛的领域调查结果图，如下所示：




我们看到，Go 应用的前 4 个领域中，有两个都是 Web 服务相关的。一个是排在第一位的 API/RPC 服务，另一个是排在第四位的 Web 服务（返回 html 页面）。考虑到后续你把

Go 应用于 Web 服务领域的机会比较大，所以，在这节课我们就选择一个 Web 服务项目作为实战小项目。

不过在真正开始我们的实战小项目前，我们先来预热一下，做一下技术铺垫。我先来给你演示一下**在 Go 中创建一个基于 HTTP 协议的 Web 服务是多么的简单**。

这种简单又要归功于 Go“面向工程”特性。在 02 讲介绍 Go 的设计哲学时，我们也说过，Go“面向工程”的特性，不仅体现在语言设计方面时刻考虑开发人员的体验，而且它还提供了完善的工具链和“自带电池”的标准库，这就使得 Go 程序大大减少了对外部第三方包的依赖。以开发 Web 服务为例，我们可以基于 Go 标准库提供的 net/http 包，轻松构建一个承载 Web 内容传输的 HTTP 服务。

下面，我们就来构建一个最简单的 HTTP 服务，这个服务的功能很简单，就是当收到一个 HTTP 请求后，给请求方返回包含“hello, world”数据的响应。

我们首先按下面步骤建立一个 simple-http-server 目录，并创建一个名为 simple-http-server 的 Go Module：

复制代码

```
$mkdir simple-http-server
$cd simple-http-server
$go mod init simple-http-server
```

由于这个 HTTP 服务比较简单，我们采用最简项目布局，也就是在 simple-http-server 目录下创建一个 main.go 源文件：

复制代码

```
package main

import "net/http"

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request){
        w.Write([]byte("hello, world"))
    })
    http.ListenAndServe(":8080", nil)
}
```

这些代码就是一个最简单的 HTTP 服务的实现了。在这个实现中，我们只使用了 Go 标准库的 http 包。可能你现在对 http 包还不熟悉，但没有关系，你现在只需要大致了解上面代码的结构与原理就可以了。

这段代码里，你要注意两个重要的函数，一个是 ListenAndServe，另一个是 HandleFunc。

你会看到，代码的第 9 行，我们通过 http 包提供的 ListenAndServe 函数，建立起一个 HTTP 服务，这个服务监听本地的 8080 端口。客户端通过这个端口与服务建立连接，发送 HTTP 请求就可以得到相应的响应结果。

那么服务端是如何处理客户端发送的请求的呢？我们看上面代码中的第 6 行。在这一行中，我们为这个服务设置了一个处理函数。这个函数的函数原型是这样的：

复制代码

```
1 func(w http.ResponseWriter, r *http.Request)
```

这个函数里有两个参数，w 和 r。第二个参数 r 代表来自客户端的 HTTP 请求，第一个参数 w 则是用来操作返回给客户端的应答的，基于 http 包实现的 HTTP 服务的处理函数都要符合这一原型。

你也发现了，在这个例子中，所有来自客户端的请求，无论请求的 URI 路径（RequestURI）是什么，请求都会被我们设置的处理函数处理。为什么会这样呢？

这是因为，我们通过 http.HandleFunc 设置这个处理函数时，传入的模式字符串为“/”。HTTP 服务器在收到请求后，会将请求中的 URI 路径与设置的模式字符串进行**最长前缀匹配**，并执行匹配到的模式字符串所对应的处理函数。在这个例子中，我们仅设置了“/”这一个模式字符串，并且所有请求的 URI 都能与之匹配，自然所有请求都会被我们设置的处理函数处理。

接着，我们再来编译运行一下这个程序，直观感受一下 HTTP 服务处理请求的过程。我们首先按下面步骤来编译并运行这个程序：

复制代码

```
$cd simple-http-server
$go build
$./simple-http-server
```

接下来，我们用 curl 命令行工具模拟客户端，向上述服务建立连接并发送 http 请求：

复制代码

```
$curl localhost:8080/
hello, world
```

我们看到，curl 成功得到了 http 服务返回的“hello, world”响应数据。到此，我们的 HTTP 服务就构建成功了。

当然了，真实世界的 Web 服务不可能像上述例子这么简单，这仅仅是一个“预热”。我想让你知道，使用 Go 构建 Web 服务是非常容易的。并且，这样的预热也能让你初步了解实现代码的结构，先有一个技术铺垫。

下面我们就进入这节课的实战小项目，一个更接近于真实世界情况的**图书管理 API 服务**。

## 图书管理 API 服务

首先，我们先来明确一下我们的业务逻辑。

在这个实战小项目中，我们模拟的是真实世界的一个书店的图书管理后端服务。这个服务为平台前端以及其他客户端，提供针对图书的 CRUD（创建、检索、更新与删除）的基于 HTTP 协议的 API。API 采用典型的 Restful 风格设计，这个服务提供的 API 集合如下：

| HTTP方法 | 请求路径（URI） | API含义 |
| --- | --- | --- |
| POST | /book | 创建一个新图书条目 |
| POST | /book/<id> | 更新一个图书条目 |
| GET | /book/<id> | 返回特定图书条目的信息 |
| GET | /book | 返回所有图书条目的信息 |
| DELETE | /book/<id> | 删除特定图书条目 |

极客时间

这个 API 服务的逻辑并不复杂。简单来说，我们通过 id 来唯一标识一本书，对于图书来说，这个 id 通常是 ISBN 号。至于客户端和服务端中请求与响应的数据，我们采用放在 HTTP 协议包体（Body）中的 Json 格式数据来承载。

业务逻辑是不是很简单啊？下面我们就直接开始创建这个项目。

## 项目建立与布局设计

我们按照下面步骤创建一个名为 bookstore 的 Go 项目并创建对应的 Go Module：

复制代码

```
$mkdir bookstore
$cd bookstore
$go mod init bookstore
go: creating new go.mod: module bookstore
```

通过上面的业务逻辑说明，我们可以把这个服务大体拆分为两大部分，一部分是 HTTP 服务器，用来对外提供 API 服务；另一部分是图书数据的存储模块，所有的图书数据均存储在这里。

同时，这是一个以构建可执行程序为目的的 Go 项目，我们参考 Go 项目布局标准一讲中的项目布局，把这个项目的结构布局设计成这样：

复制代码

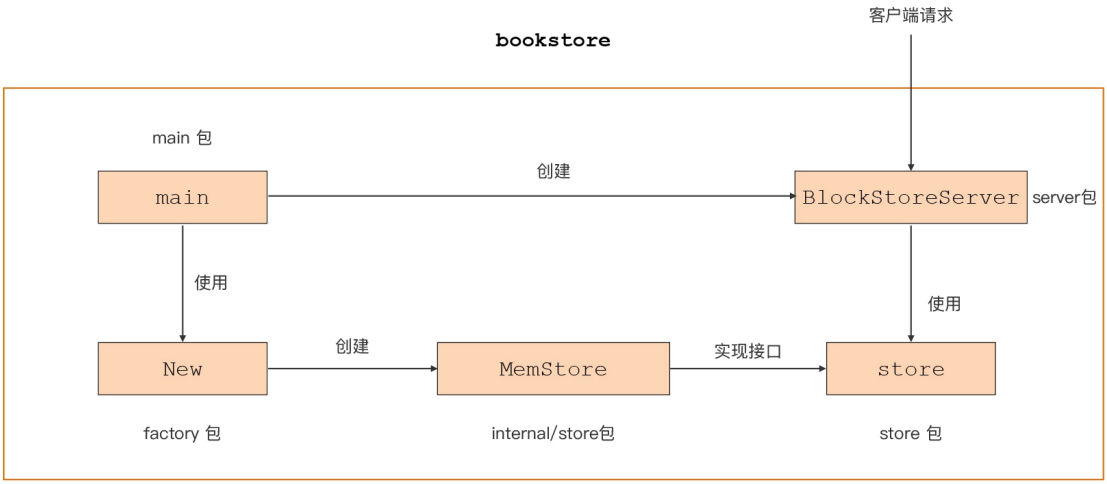
```
├── cmd/
│   └── bookstore/          // 放置bookstore main包源码
```

```
3  │       └── main.go
4  ├── go.mod                  // module bookstore的go.mod
5  ├── go.sum
6  ├── internal/               // 存放项目内部包的目录
7  │   └── store/
8  │       └── memstore.go
9  ├── server/                 // HTTP服务器模块
10 │   ├── middleware/
11 │   │   └── middleware.go
12 │   └── server.go
13 └── store/                  // 图书数据存储模块
14     ├── factory/
15     │   └── factory.go
16     └── store.go
```

现在，我们既给出了这个项目的结构布局，也给出了这个项目最终实现的源码文件分布情况。下面我们就从 main 包开始，自上而下逐一看看这个项目的模块设计与实现。

## 项目 main 包

main 包是主要包，为了搞清楚各个模块之间的关系，我在这里给出了 main 包的实现逻辑图：




极客时间

同时，我也列出了 main 包（main.go）的所有代码，你可以先花几分钟看一下：

复制代码

```
package main

import (
    _ "bookstore/internal/store"
    "bookstore/server"
    "bookstore/store/factory"
    "context"
    "log"
    "os"
    "os/signal"
    "syscall"
    "time"
)

func main() {
    s, err := factory.New("mem") // 创建图书数据存储模块实例
    if err != nil {
        panic(err)
    }

    srv := server.NewBookStoreServer(":8080", s) // 创建http服务实例

    errChan, err := srv.ListenAndServe() // 运行http服务
    if err != nil {
        log.Println("web server start failed:", err)
        return
    }
    log.Println("web server start ok")

    c := make(chan os.Signal, 1)
    signal.Notify(c, syscall.SIGINT, syscall.SIGTERM)

    select { // 监视来自errChan以及c的事件
    case err = <-errChan:
        log.Println("web server run failed:", err)
        return
    case <-c:
        log.Println("bookstore program is exiting...")
        ctx, cf := context.WithTimeout(context.Background(), time.Second)
        defer cf()
        err = srv.Shutdown(ctx) // 优雅关闭http服务实例
    }

    if err != nil {
        log.Println("bookstore program exit error:", err)
        return
    }
    log.Println("bookstore program exit ok")
}
```

在 Go 中，main 包不仅包含了整个程序的入口，它还是整个程序中主要模块初始化与组装的场所。那对应在我们这个程序中，主要模块就是第 16 行的创建图书存储模块实例，以及第 21 行创建 HTTP 服务模块实例。而且，你还要注意的是，第 21 行创建 HTTP 服务模块实例的时候，我们把图书数据存储实例 s 作为参数，传递给了 NewBookStoreServer 函数。这两个实例的创建原理，我们等会再来细细探讨。

这里，我们重点来看 main 函数的后半部分（第 30 行~ 第 42 行），这里表示的是，我们通过监视系统信号实现了 http 服务实例的优雅退出。

所谓优雅退出，指的就是程序有机会等待其他的事情处理完再退出。比如尚未完成的事务处理、清理资源（比如关闭文件描述符、关闭 socket）、保存必要中间状态、内存数据持久化落盘，等等。如果你经常用 Go 来编写 http 服务，那么 http 服务如何优雅退出，就是你经常要考虑的问题。

在这个问题的具体实现上，我们通过 signal 包的 Notify 捕获了 SIGINT、SIGTERM 这两个系统信号。这样，当这两个信号中的任何一个触发时，我们的 http 服务实例都有机会在退出前做一些清理工作。

然后，我们再使用 http 服务实例（srv）自身提供的 Shutdown 方法，来实现 http 服务实例内部的退出清理工作，包括：立即关闭所有 listener、关闭所有空闲的连接、等待处于活动状态的连接处理完毕，等等。当 http 服务实例的清理工作完成后，我们整个程序就可以正常退出了。

接下来，我们再重点看看构成 bookstore 程序的两个主要模块：图书数据存储模块与 HTTP 服务模块的实现。我们按照 main 函数中的初始化顺序，先来看看图书数据存储模块。

## 图书数据存储模块（store）

图书数据存储模块的职责很清晰，就是用来**存储整个 bookstore 的图书数据**的。图书数据存储有很多种实现方式，最简单的方式莫过于在内存中创建一个 map，以图书 id 作为 key，来保存图书信息，我们在这一讲中也会采用这种方式。但如果我们要考虑上生产环境，数据要进行持久化，那么最实际的方式就是通过 Nosql 数据库甚至是关系型数据库，实现对图书数据的存储与管理。

考虑到对多种存储实现方式的支持，我们将针对图书的有限种存储操作，放置在一个接口类型 Store 中，如下源码所示：

复制代码

```
// store/store.go

type Book struct {
    Id      string   `json:"id"`       // 图书ISBN ID
    Name    string   `json:"name"`     // 图书名称
    Authors []string `json:"authors"` // 图书作者
    Press   string   `json:"press"`    // 出版社
}

type Store interface {
    Create(*Book) error         // 创建一个新图书条目
    Update(*Book) error         // 更新某图书条目
    Get(string) (Book, error)   // 获取某图书信息
    GetAll() ([]Book, error)    // 获取所有图书信息
    Delete(string) error        // 删除某图书条目
}
```

这里，我们建立了一个对应图书条目的抽象数据类型 Book，以及针对 Book 存取的接口类型 Store。这样，对于想要进行图书数据操作的一方来说，他只需要得到一个满足 Store 接口的实例，就可以实现对图书数据的存储操作了，不用再关心图书数据究竟采用了何种存储方式。这就实现了图书存储操作与底层图书数据存储方式的解耦。而且，这种面向接口编程也是 Go 组合设计哲学的一个重要体现。

那我们具体如何创建一个满足 Store 接口的实例呢？我们可以参考《设计模式》提供的多种创建型模式，选择一种 Go 风格的工厂模式（创建型模式的一种）来实现满足 Store 接口实例的创建。我们看一下 store/factory 包的源码：

复制代码

```
// store/factory/factory.go

var (
    providersMu sync.RWMutex
    providers   = make(map[string]store.Store)
)

func Register(name string, p store.Store) {
    providersMu.Lock()
   defer providersMu.Unlock()
```

```
    if p == nil {
        panic("store: Register provider is nil")
    }

    if _, dup := providers[name]; dup {
        panic("store: Register called twice for provider " + name)
    }
    providers[name] = p
}

func New(providerName string) (store.Store, error) {
    providersMu.RLock()
    p, ok := providers[providerName]
    providersMu.RUnlock()
    if !ok {
        return nil, fmt.Errorf("store: unknown provider %s", providerName)
    }

    return p, nil
}
```

这段代码实际上是效仿了 Go 标准库的 database/sql 包采用的方式，factory 包采用了一个 map 类型数据，对工厂可以“生产”的、满足 Store 接口的实例类型进行管理。factory 包还提供了 Register 函数，让各个实现 Store 接口的类型可以把自己“注册”到工厂中来。

一旦注册成功，factory 包就可以“生产”出这种满足 Store 接口的类型实例。而依赖 Store 接口的使用方，只需要调用 factory 包的 New 函数，再传入期望使用的图书存储实现的名称，就可以得到对应的类型实例了。

在项目的 internal/store 目录下，我们还提供了一个基于内存 map 的 Store 接口的实现，我们具体看一下这个实现是怎么自注册到 factory 包中的：

复制代码

```
// internal/store/memstore.go

package store

import (
    mystore "bookstore/store"
    factory "bookstore/store/factory"
    "sync"
)

```

```
func init() {
    factory.Register("mem", &MemStore{
        books: make(map[string]*mystore.Book),
    })
}

type MemStore struct {
    sync.RWMutex
    books map[string]*mystore.Book
}
```

从 memstore 的代码来看，它是在包的 init 函数中调用 factory 包提供的 Register 函数，把自己的实例以“mem”的名称注册到 factory 中的。这样做有一个好处，依赖 Store 接口进行图书数据管理的一方，只要导入 internal/store 这个包，就可以自动完成注册动作了。

理解了这个之后，我们再看下面 main 包中，创建图书数据存储模块实例时采用的代码，是不是就豁然开朗了？

复制代码

```
import (
    ... ...
    _ "bookstore/internal/store" // internal/store将自身注册到factory中
)

func main() {
    s, err := factory.New("mem") // 创建名为"mem"的图书数据存储模块实例
    if err != nil {
        panic(err)
    }
    ... ...
}
```

至于 memstore.go 中图书数据存储的具体逻辑，就比较简单了，我这里就不详细分析了，你课后自己阅读一下吧。

接着，我们再来看看 bookstore 程序的另外一个重要模块：HTTP 服务模块。

## HTTP 服务模块（server）

HTTP 服务模块的职责是**对外提供 HTTP API 服务，处理来自客户端的各种请求，并通过 Store 接口实例执行针对图书数据的相关操作**。这里，我们抽象处理一个 server 包，这个包中定义了一个 BookStoreServer 类型如下：

复制代码

```
// server/server.go

type BookStoreServer struct {
    s   store.Store
    srv *http.Server
}
```

我们看到，这个类型实质上就是一个标准库的 http.Server，并且组合了来自 store.Store 接口的能力。server 包提供了 NewBookStoreServer 函数，用来创建一个 BookStoreServer 类型实例：

复制代码

```
// server/server.go

func NewBookStoreServer(addr string, s store.Store) *BookStoreServer {
    srv := &BookStoreServer{
        s: s,
        srv: &http.Server{
            Addr: addr,
        },
    }

    router := mux.NewRouter()
    router.HandleFunc("/book", srv.createBookHandler).Methods("POST")
    router.HandleFunc("/book/{id}", srv.updateBookHandler).Methods("POST")
    router.HandleFunc("/book/{id}", srv.getBookHandler).Methods("GET")
    router.HandleFunc("/book", srv.getAllBooksHandler).Methods("GET")
    router.HandleFunc("/book/{id}", srv.delBookHandler).Methods("DELETE")

    srv.srv.Handler = middleware.Logging(middleware.Validating(router))
    return srv
}
```

我们看到函数 NewBookStoreServer 接受两个参数，一个是 HTTP 服务监听的服务地址，另外一个是实现了 store.Store 接口的类型实例。这种函数原型的设计是 Go 语言的一

种惯用设计方法，也就是接受一个接口类型参数，返回一个具体类型。返回的具体类型组合了传入的接口类型的能力。

这个时候，和前面预热时实现的简单 http 服务一样，我们还需为 HTTP 服务器设置请求的处理函数。

由于这个服务请求 URI 的模式字符串比较复杂，标准库 http 包内置的 URI 路径模式匹配器（ServeMux，也称为路由管理器）不能满足我们的需求，因此在这里，我们需要借助一个第三方包 github.com/gorilla/mux 来实现我们的需求。

在上面代码的第 11 行到第 16 行，我们针对不同 URI 路径模式设置了不同的处理函数。我们以 createBookHandler 和 getBookHandler 为例来看看这些处理函数的实现：

复制代码

```
1  // server/server.go
2
3    func (bs *BookStoreServer) createBookHandler(w http.ResponseWriter, req *htt
4        dec := json.NewDecoder(req.Body)
5        var book store.Book
6        if err := dec.Decode(&book); err != nil {
7            http.Error(w, err.Error(), http.StatusBadRequest)
8            return
9        }
10
11       if err := bs.s.Create(&book); err != nil {
12           http.Error(w, err.Error(), http.StatusBadRequest)
13           return
14       }
15   }
16
17   func (bs *BookStoreServer) getBookHandler(w http.ResponseWriter, req *http.R
18       id, ok := mux.Vars(req)["id"]
19       if !ok {
20           http.Error(w, "no id found in request", http.StatusBadRequest)
21           return
22       }
23
24      book, err := bs.s.Get(id)
25       if err != nil {
26          http.Error(w, err.Error(), http.StatusBadRequest)
27          return
28       }
29
30       response(w, book)
31   }
```

```
32
33  func response(w http.ResponseWriter, v interface{}) {
34      data, err := json.Marshal(v)
35      if err != nil {
36          http.Error(w, err.Error(), http.StatusInternalServerError)
37          return
38      }
39      w.Header().Set("Content-Type", "application/json")
40      w.Write(data)
41  }
```

这些处理函数的实现都大同小异，都是先获取 http 请求包体数据，然后通过标准库 json 包将这些数据，解码（decode）为我们需要的 store.Book 结构体实例，再通过 Store 接口对图书数据进行存储操作。如果我们是获取图书数据的请求，那么处理函数将通过 response 函数，把取出的图书数据编码到 http 响应的包体中，并返回给客户端。

然后，在 NewBookStoreServer 函数实现的尾部，我们还看到了这样一行代码：

复制代码

```
1       srv.srv.Handler = middleware.Logging(middleware.Validating(router))
```

这行代码的意思是说，我们在 router 的外围包裹了两层 middleware。什么是 middleware 呢？对于我们的上下文来说，这些 middleware 就是一些通用的 http 处理函数。我们看一下这里的两个 middleware，也就是 Logging 与 Validating 函数的实现：

复制代码

```
1  // server/middleware/middleware.go
2
3   func Logging(next http.Handler) http.Handler {
4      return http.HandlerFunc(func(w http.ResponseWriter, req *http.Request) {
5          log.Printf("recv a %s request from %s", req.Method, req.RemoteAddr)
6          next.ServeHTTP(w, req)
7      })
8  }
9
10 func Validating(next http.Handler) http.Handler {
11     return http.HandlerFunc(func(w http.ResponseWriter, req *http.Request) {
12         contentType := req.Header.Get("Content-Type")
13         mediatype, _, err := mime.ParseMediaType(contentType)
14         if err != nil {
15             http.Error(w, err.Error(), http.StatusBadRequest)
16             return
```

```
        }
        if mediatype != "application/json" {
            http.Error(w, "invalid Content-Type", http.StatusUnsupportedMedia
            return
        }
        next.ServeHTTP(w, req)
    })
}
```

我们看到，Logging 函数主要用来输出每个到达的 HTTP 请求的一些概要信息，而 Validating 则会对每个 http 请求的头部进行检查，检查 Content-Type 头字段所表示的媒体类型是否为 application/json。这些通用的 middleware 函数，会被串联到每个真正的处理函数之前，避免我们在每个处理函数中重复实现这些逻辑。

创建完 BookStoreServer 实例后，我们就可以调用其 ListenAndServe 方法运行这个 http 服务了，显然这个方法的名字是仿效 http.Server 类型的同名方法，我们的实现是这样的：

复制代码

```
// server/server.go

 func (bs *BookStoreServer) ListenAndServe() (<-chan error, error) {
     var err error
     errChan := make(chan error)
     go func() {
         err = bs.srv.ListenAndServe()
         errChan <- err
     }()

     select {
     case err = <-errChan:
         return nil, err
     case <-time.After(time.Second):
         return errChan, nil
     }
}
```

我们看到，这个函数把 BookStoreServer 内部的 http.Server 的运行，放置到一个单独的轻量级线程 Goroutine 中。这是因为，http.Server.ListenAndServe 会阻塞代码的继续运行，如果不把它放在单独的 Goroutine 中，后面的代码将无法得到执行。

为了检测到 http.Server.ListenAndServe 的运行状态，我们再通过一个 channel（位于第 5 行的 errChan），在新创建的 Goroutine 与主 Goroutine 之间建立的通信渠道。通过这个渠道，这样我们能及时得到 http server 的运行状态。

## 编译、运行与验证

到这里，bookstore 项目的大部分重要代码我们都分析了一遍，是时候将程序跑起来看看了。

不过，因为我们在程序中引入了一个第三方依赖包，所以在构建项目之前，我们需要执行下面这个命令，让 Go 命令自动分析依赖项和版本，并更新 go.mod：

```
$go mod tidy
go: finding module for package github.com/gorilla/mux
go: found github.com/gorilla/mux in github.com/gorilla/mux v1.8.0
```

完成后，我们就可以按下面的步骤来构建并执行 bookstore 了：

```
$go build bookstore/cmd/bookstore
$./bookstore
2021/10/05 16:08:36 web server start ok
```

如果你看到上面这个输出的日志，说明我们的程序启动成功了。

现在，我们就可以像前面一样使用 curl 命令行工具，模仿客户端向 bookstore 服务发起请求了，比如创建一个新书条目：

```
$curl -X POST -H "Content-Type:application/json" -d '{"id": "978-7-111-55842-2
```

此时服务端会输出如下日志，表明我们的 bookstore 服务收到了客户端请求。

```
1 2021/10/05 16:09:10 recv a POST request from [::1]:58021
```

复制代码

接下来，我们再来获取一下这本书的信息：

复制代码

```
1 $curl -X GET -H "Content-Type:application/json" localhost:8080/book/978-7-111-
2 {"id":"978-7-111-55842-2","name":"The Go Programming Language","authors":["Ala
```

我们看到 curl 得到的响应与我们预期的是一致的。

好了，我们不再进一步验证了，你课后还可以自行编译、执行并验证。

## 小结

到这里，我们就完成了我们第一个实战小项目，不知道你感觉如何呢？

在这一讲中，我们带你用 Go 语言构建了一个最简单的 HTTP 服务，以及一个接近真实的图书管理 API 服务。在整个实战小项目的实现过程中，你也能初步学习到 Go 编码时常用的一些惯用法，比如基于接口的组合、类似 database/sql 所使用的惯用创建模式，等等。

通过这节课的学习，你是否体会到了 Go 语言的魅力了呢？是否察觉到 Go 编码与其他主流语言不同的风格了呢？其实不论你的理解程度有多少，都不重要。只要你能“照猫画虎”地将上面的程序自己编写一遍，构建、运行起来并验证一遍，就算是完美达成这一讲的目标了。

你在这个过程肯定会有各种各样的问题，但没关系，这些问题会成为你继续向下学习 Go 的动力。毕竟，**带着问题的学习，能让你的学习过程更有的放矢、更高效**。

## 思考题

如果你完成了今天的代码，觉得自己学有余力，可以再挑战一下，不妨试试基于 nosql 数据库，我们怎么实现一个新 store.Store 接口的实现吧？

欢迎把这节课分享给更多对 Go 语言感兴趣的朋友。我是 Tony Bai，我们下节课见。

## 资源链接

这节课的图书管理项目的完整源码在这里！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 4　提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　08 | 入口函数与包初始化：搞清Go程序的执行次序

下一篇　10 | 变量声明：静态语言有别于动态语言的重要特征

## 精选留言 (22)

写留言

**自由**
2021-11-05

Tony Bai 老师的这篇“简单”的 Web 代码构建，细节特别多，足矣看出功力深厚。例如锁的选型、工程化、为什么用锁、面向接口、设计模式、通过 Channel 通信而不是共享内存、panic 和 error 的区别。我认为这样的代码，是一个完美的开始。

展开

1 3

**布凡**

2021-11-04

终于实验成功：
以下几个点需注意：
1、项目应放到 gopath货goroot相关的目录下，否则本地包的引用会报错，报错信息如下："could not import errors (cannot find package "errors" in any of c:\go\src\errors (from $GOROOT)...)"...

展开

2

**return**

2021-11-01

老师讲的太赞了
市面上 讲 go 接口思想很好， 推荐组合思想。 但是都光说不练，看懂但学不会。
今天老师 给的这个例子 真实，简短 且 把接口和组合的 思想都 做了出来， 太牛了。
这一篇 需要实践而且需要反复揣摩。

展开

1 2

**lesserror**

2021-11-02

感谢 Tony Bai 老师这样由浅入深，并且尽可能贴近实战的讲解。

有以下困惑，麻烦解忧。

1. 对于一个module中，相同的包名，go内部是根据文件位置的不同，加以区分的吧？ ...

展开

作者回复: 问题1. 你的理解没有错。
问题2. 可以不用另起，记得好像有些工具在格式化时会自动填上。
3. 定位不同，这个是入门专栏。那个进阶专栏。有很小部分有相似的地方，但讲解方式都做了调整和优化。建议学完这个专栏后，再去看看那个专栏。
4. rpc封装的很深，如果仅是使用的话，没啥东西。这个专栏更多聚焦于go语法以及尽量使用标准库的东西做一些实用程序和项目。

2 1

**猫饼**
2021-11-02

我果然还是太菜了 我开始看不懂了

展开

作者回复: 先“照猫画虎”吧。一旦涉及此类实战小项目，必然会有没有系统学习的内容。后面学习新语法时，可以回过头来复习。

1 1

**高雪斌**
2021-11-01

C++转Go, Go语言设计的确实很简洁、精妙。Tony Bai老师从语言结构开始讲的风格非常受用，比通常的从语法开始讲的形式要好，虽然一些代码现在看不明白，但这样带着问题学习语法更容易理解。

1

**jc9090kkk**
2021-11-01

这节课的项目内容理解起来不困难，但是对于实际的生产项目而言，尤其是对第三方的中间件有依赖的前提下，大都会针对的将配置单独存储为配置文件以方便维护，我想问下老师go项目针对配置文件有什么最佳实践方式吗? 我自己本地写了一个小项目，用的是以下方式来配置MySQL连接的，我总觉得不太优雅，大多项目会根据开发环境的不同选取不同的配置文件作为配置项加载，但是如果通过硬编码的方式添加进去会让项目变得很奇...

展开

作者回复: 生产项目一般不会将配置“硬编码”进去的。一般而言，配置项会存储在配置文件中，也有通过命令行参数传入程序的，也可以通过环境变量传入程序。Go标准库没有内置配置读写框架，目前go社区应用较多的第三方库是Go核心团队成员开发的viper(github.com/spf13/viper)。

对于一些更大的平台，常常有很多服务，这些服务的配置一般存储在专门的配置中心中，由配置中心管理与分发。

4 1

**光**

2021-11-05

memorystore.go
 if book.Authors != nil {
        nBook.Authors = book.Authors
    }
这里是不是应该改成：...

展开

1

**snow**
2021-11-05

我看这里使用了mux包，我只用过gin包，请问这两个老师更喜欢哪个？以及这里选择mux的原因。

**胖郃**
2021-11-05

老师，采用Restful规范，更新一条图书条目 http方法采用PUT，当然post也是可以的，put book/id，是我太限定自己了吗？

展开



**小鬼**
2021-11-05

go mod tidy
go: finding module for package github.com/gorilla/mux
bookstore/server imports
        github.com/gorilla/mux: module github.com/gorilla/mux: Get "https://proxy.golang.org/github.com/gorilla/mux/@v/list": dial tcp...

展开

1



**Dunbreak**
2021-11-04

请问一下，为什么我按照源码写了这个小项目，但是factory包的Register函数从来都不调用呢？断点加在memstore.go的 init函数中也不执行，最终导致run起来直接就报panic了

2



**Geek_8ed998**
2021-11-04

package bookstore/cmd/bookstore is not in GOROOT (/usr/local/go/src/bookstore/cmd/bookstore) 这个啥原因阿

展开

1

**不负青春不负己✌**

2021-11-02

突然发现一个好玩的github 功能，即将.com 变成dev 会变成 web 版的 https://github.dev/bigwhite/publication/tree/master/column/timegeek/go-first-course/09/bookstore

1

**不负青春不负己✌**

2021-11-02

这节有点复杂，先run吧，后面在复习一下:)

展开

作者回复: 不明白就反复多看多听几遍。有问题随时留言。

**进化菌**

2021-11-02

开始感觉有些难度了，得多看两遍，把程序跑起来才行了~

**Guoo**

2021-11-02

老师确实过来人，正准备找个项目手撸一遍呢

展开

1

**陈东**

2021-11-01

先把代码敲一遍,捋顺之后,再理解,是不是对新手是比较好的方式?

作者回复: 学习任何一门编程语言，这个过程都是必不可少的。只是对着专栏文章或书籍看，那是学不会编码的，必须动手。

**张鹏**

2021-11-01

老师，Register(name string, p store.Store)方法第二参数要求传入Store，memstore.go中的init方法使用Register时传入的是MemStore类型，类型不匹配吧？

作者回复: store.Store是一个接口类型，而MemStore实现了这个接口，自然是可以传给store.Store的。关于接口类型，在后续课程中会详细讲解。这里大家先有个感性认识吧。

3

**张鹏**

2021-11-01

老师，你每个文件的包名怎么命名？根据目录来吗？

展开

作者回复: 包名可以任意命名，没有限制，但社区公认的好的包名通常为一个小写单词。一个目录下仅允许存放一个包。通常一个优秀的实践是包名与目录名相同。但也有很多项目没有遵守这个约定俗成的规则。

1